Chapter 2

Paradigm Examples – Sorting

Our purpose in this chapter is to illustrate how the Leda programming language supports each of the four different programming paradigms discussed in Chapter 1. We will do this by examining one specific problem, an insertion sort performed using linked-list structures, and provide solutions to this problem written in Leda that exemplify the salient characteristics of each of the four paradigms.

2.1 Linked Lists

A linked list is commonly defined as series of link values, which are each dynamically constructed as instances of a structure. Although Leda does not have structures in the conventional sense (as found in, for example, C or Pascal), an adequate substitute is provided by a class declaration that defines only data fields, and does not define any of its own behavior. Such a declaration is shown in Figure 2.1. Each instance of the class maintains a single integer value, and a value which represents the remainder of the list.[1] A list of integers is simply a collection of such values, each element of the list referring to the next, with the final link holding a "next" field which is uninitialized.

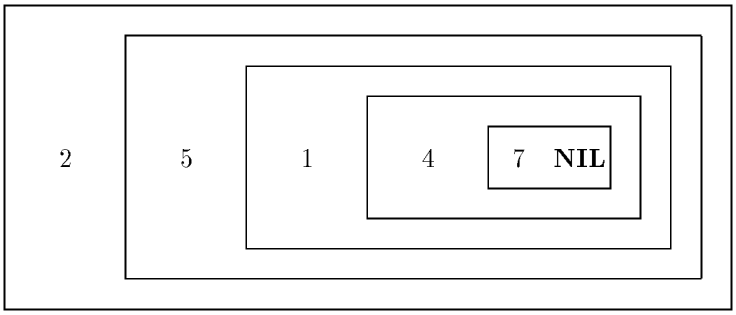
Note that Leda does not have the concept of an explicit "pointer". Instead, the Leda language uses the more abstract notion that a value may or may not be initialized. Each element in the list structure holds, in addition to an integer value, a field which is, in fact, the remainder of the list. In the final element this value has not been initialized by the programmer, and thus retains the value assigned by the language to all newly created variables, which is represented by the constant `NIL`. The programmer can think of this state

[1] This list type can only maintain integer values. In the next chapter we will introduce a feature of the Leda language that permits the creation of datatypes that can hold arbitrary types of values; our purpose here is to simply provide a gentle introduction to both Leda and each of the different programming styles.

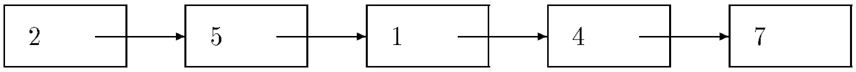
```
class intList;
var
    value : integer;
    next : intList;
end;

function head (lst : intList)->integer;
begin       { return first element in list }
    return lst.value;
end;

function tail (lst : intList)->intList;
begin       { return list with first element removed }
    return lst.next;
end;

function empty (lst : intList)->boolean;
begin       { uninitialized list is empty, all other lists are non-empty }
    return ~ defined(lst);
end;
```

Figure 2.1: Declaration of a Link for Linked Lists

as representing *undefined*, or *uninitialized*.[2] Whether or not a variable holds the value `NIL` can be tested using a system-provided predicate named `defined`. A realistic picture of a linked list is therefore a series of nested boxes:



As this method of representation can quickly become unwieldy, we can instead represent lists in the more conventional fashion, where the link field in each element is written as an arrow:



We will use four functions in the manipulation of our simple linked lists. Every class declaration automatically causes the generation of an associated *constructor* function, having the same name as the class type. The constructor function takes a series of arguments matching in type and order the data fields in the class declaration. The constructor provides for both creation and initialization of a new instance of the class. The linked list of five elements described above might therefore be generated in the following fashion:

```
var
    aList : intList;
begin
    aList := intList(2, intList(5, intList(1,
            intList(4, intList(7, NIL)))));
    ...
end;
```

The value `NIL` is a polymorphic value, which means it can be assigned to any data type.

While a constructor is used to put a list together, the functions `head` and `tail` (Figure 2.1) are used to break a list apart. The function `head` returns the value assigned with the first element in a list, while the function `tail` returns the list formed with the first

[2]Although, in fact, a variable holding the value `NIL` is both well defined and initialized. It simply has not been reassigned to any new value by the programmer.

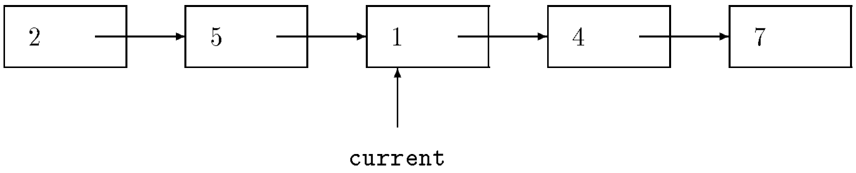
element removed. Both functions will generate a run-time error if they are applied to an empty list. Finally, the boolean function `empty` returns true if the argument is an empty list, and false otherwise.

2.2 An Imperative Sorting Procedure

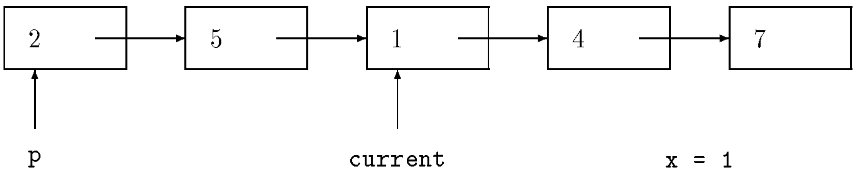
The defining characteristic of the imperative approach to problem solving is the direct manipulation and modification of structures which maintain the "state" of the calculation. This is exemplified by the sorting procedure shown in Figure 2.2. The procedure takes as a single argument a list, which is transmitted using pass-by-reference (indicated by the `byRef` designation) and modified in situ.

Because the imperative paradigm emphasizes changes to state, imperative programs are often best understood through the use of invariants. An invariant is nothing more than a description of some aspect of the state of memory when execution reaches a given point in the course of execution. Generally invariants provide a way of describing the portion of a problem that has already been solved; and thus, indirectly, the portion that remains to be solved. While invariants are sometimes cloaked in dense formalisms, for our purposes (and, generally, for most purposes) a quite informal approach is sufficient.

The imperative procedure operates by sorting the initial prefix of the list, enlarging the ordered section, element by element, until finally the entire list is sorted. The variable `current` is used to loop over the linked structure, preserving the invariant that all elements prior to the current are properly ordered. This invariant is trivial to initially establish; when `current` points to the first element, the list of previous values is empty, and thus automatically ordered. To see how the invariant is established in general, consider the following picture, in which the first two elements have been already ordered, and the procedure is about to place the third element into sequence.



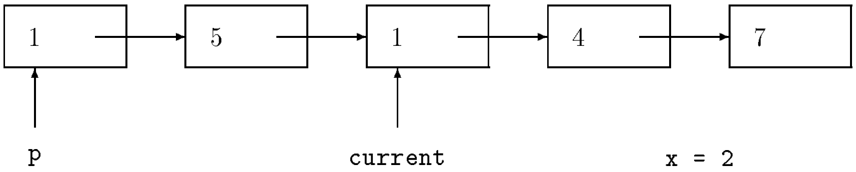
The value of the current link is placed into a variable named `x`. A second pointer, named `p`, is then used to loop over each link prior to the current.



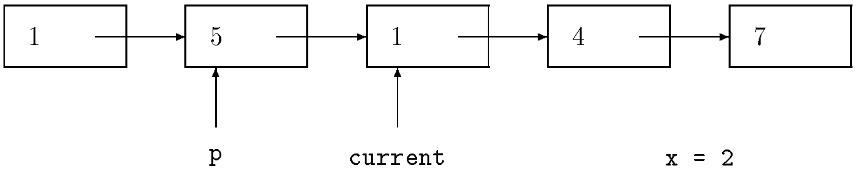
```
function sort (byRef aList : intList);
var
    current, p : intList;
    x, temp : integer;
begin
    current := aList;
    while defined(current) do begin

        { place current element into proper location }

        x := current.value;
        p := aList;

        while p ~= current do begin
            if x < p.value then begin
                temp := p.value;
                p.value := x;
                x := temp;
            end;
            p := p.next;
        end;

        { advance to next position }
        current.value := x;
        current := current.next;
    end;
end;
```

Figure 2.2: The Imperative Sorting Procedure

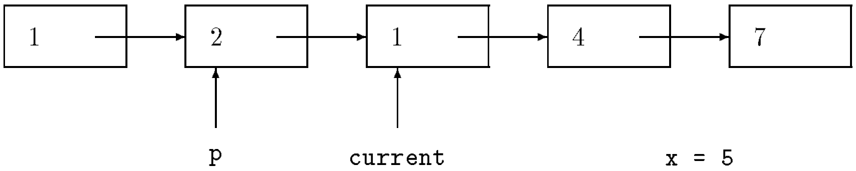
If the value pointed to by `p` is less than the value in `x`, they are swapped.



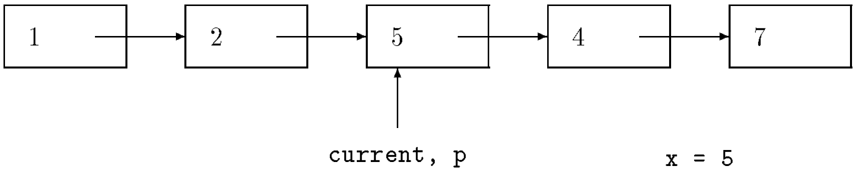
The pointer `p` is then incremented to the next element.



Once again the value of the node pointed to by `p` is compared to the variable `x`. If `x` is smaller they are swapped.



This continues until the pointer `p` reaches `current`, which causes the loop to terminate. Note that the ~= operator is testing for object non-identity (that is, "does not represent the same object") as opposed to non-equality, which is the <> operator. The latter is often redefined for different data-types (numbers, strings, user-defined types). When the loop terminates, the final value maintained in the variable `x` (which must be the largest value seen so far) is then assigned to the current link in the list.



The invariant has therefore been strengthened to say that the entire list up to *and including* the current element has been ordered, and the pointer to the current element can therefore be advanced and the process will begin once again. When the entire list has finally been examined, it will have been sorted.

To reiterate the point we made at the beginning of this section, the fundamental property of this solution that makes it characteristic of imperative programming is the idea of computing by making small changes to a data structure, until by a long series of such small changes the desired outcome is reached.

2.3 A Logic Programming Sorting Procedure

If imperative program is concerned with the development of a series of detailed step-by-step instructions which when executed combine together to produce a desired outcome, then the antithesis of the approach is perhaps epitomized by the techniques used in programming in the logic programming style. The hallmark of logic programming is that it is one of the purest examples of *declarative programming*. Rather than being concerned with *how* a solution will be attained, a logic programmer concentrates on defining precisely the *characteristics* of the desired solution, leaving it to the computer to discover exactly how this solution will be found.

Of course, the declarative nature of logic programs is in some sense a sham, a charade. If the search for a solution fits a common and well understood pattern, then the built-in facilities provided for logic programming can be employed. If a problem is slightly different, then the logic programmer must often provide "hints" or directions for the compiler in order to mold the problem into the correct format. In languages in which all computation must be expressed in a logic programming style, these non-logical directives sometimes appear to fly in the face of, and severely detract from, the declarative nature of the paradigm.

A logic programming solution to the insertion sorting problem is shown in Figure 2.3. Logic programs are built around a stereotypical type of function, which returns a datatype called a *relation*. A relation can be thought of as a form of boolean, although the reality (namely, the implementation) is slightly more complicated. The intent is that a function returning a relation is in fact returning true if the arguments satisfy a certain property. More importantly, the difference between a relation and a boolean is that a relation can be used to generate values in such a manner that the property is satisfied.

The first relation shown in Figure 2.3 is given by the function named `insertion`. This relation can be thought of as characterizing what it means for a new element (the variable `x`) to be inserted into a sorted list (the variable `startList`) resulting in a new list (the variable `newList`, which is transmitted using pass-by-reference, and thus can be used to return a value to the caller.)

The body of a relation is typically a single expression constructed using logical operators. The logical expression divides naturally into several cases. If the starting list is empty, then a new list is simply a list containing a single element. A special form of assignment is used, written using the `<-` arrow. This form of assignment performs the binding of value to name, and in addition returns a boolean "true" value. (A third characteristic of the assignment

```
function insertion (startList : intList,
    x : integer, byRef newList : intList)->relation;
var
    temp : intList;
begin
    return empty(startList) & newList <- intList(x, NIL)
        | ~ empty(startList) &
            (x < head(startList) & newList <- intList(x, startList)
            | x >= head(startList) &
                insertion(tail(startList), x, temp) &
                newList <- intList(head(startList),temp));
end;

function sort (startList:intList, byRef endList:intList)->relation;
var
    temp : intList;
begin
    return empty(startList) & endList <- startList
        | ~ empty(startList) & sort(tail(startList), temp) &
            insertion(temp, head(startList), endList);
end;
```

Figure 2.3: A Logic Programming Program for Insertion Sort

operator is the ability to "undo" the binding of name to value in certain situations. However, this property of the arrow operator is not used in the current problem.)

If, on the other hand, the starting list is not empty, then the insertion of a value which is smaller than the first element in the list is simply a new list constructed with the new element at the beginning. Finally, if the new value is larger than or equal to the start of the list then the insertion is characterized by the list that has the head of the starting list for a first element, and the insertion of the new element into the remainder of the list. (The latter requires two steps and the introduction of a temporary variable).

A sorted list can similarly be recursively characterized by a defining predicate. A list is sorted if it is empty, or if not empty if the first element is an inserted value into the sorted version of the remainder of the list.

Notice there are no imperative actions (other than assignment) anywhere in these two functions. There are only boolean expressions which define the properties that a solution must possess. Furthermore, these properties closely match a logical definition of what it means for an insertion to be performed on a list, and what it means for a list to be sorted. It is this close relationship between description or definition and program that makes logic programs particularly appealing in many problem domains.

2.3.1 Mixing Logical and Imperative Programming

The clauses in the two functions in Figure 2.3 exhibit a form that is frequently encountered in logic programs. A relation is defined by a series of disjunctions ("or" expressions) over a series of conjunctions ("and" expressions). Furthermore, the beginning conjuncts in each expression are used to characterize a special case of interest, while the remaining conjuncts define some property or perform some action specific to the case just defined.

Since each clause is logically independent of the others, one clause is frequently prefaced by the exact logical inverse of the immediately preceding clause. In effect, the initial clauses are being used to control the flow of the program. In this situation, an improvement in efficiency (and, in the eyes of some readers, an improvement in clarity) can be obtained by using explicit control flow directives in place of disjoint clauses. This is shown in Figure 2.4. Expressed in this form, the logic programming solution begins to look very similar to the functional programming solution we will discuss next.

2.4 A Functional Solution

The remaining two paradigms to be examined are the functional programming paradigm and the object-oriented paradigm. The two techniques are similar in that a broad goal for both approaches is to permit and facilitate the creation of general, reusable tools which can be employed to raise the level of abstraction in programming. That is, rather than starting each new program on a foundation composed solely of the programming language alone, as if no other program had ever been written, both the functional and object-oriented

```
function insertion (startList : intList,
    x : integer, byRef newList : intList)->relation;
var
    temp : intList;
begin
    if empty(startList) then
        return newList <- intList(x, NIL)
    else
        if x < head(startList) then
            return newList <- intList(x, startList)
        else
            return insertion(tail(startList), x, temp) &
                newList <- intList(head(startList), temp);
end;

function sort (startList:intList, byRef endList:intList)->relation;
var
    temp : intList;
begin
    if empty(startList) then
        return endList <- NIL
    else
        return sort(tail(startList), temp) &
            insertion(temp, head(startList), endList);
end;
```

Figure 2.4: Sort Implemented with Imperative and Logic Programming

programming paradigms encourage the development of tools which can be created within the language itself, but which can be carried from one project to the next.

On a superficial level, where the two techniques differ is in the mechanism used to express these tools. In functional programming such tools are expressed as functions, while in the object-oriented approach tools are generally expressed as classes. On a deeper level, these two philosophies differ in how they view the process of computation. The object-oriented approach builds on the conventional view that computation proceeds as a series of steps which each make small changes to the state of the machine. As we noted in Chapter 1, the object-oriented technique takes this to a higher level of abstraction, providing a mechanism for generating "computers in the small". Each miniaturized-computer is responsible for a certain amount of memory and a certain set of behaviors. Rather than directly manipulating memory, computation proceeds by asking different computing units in turn to manipulate their own memory.

Functional programming, on the other hand, rejects altogether the notion that computation consists of the modification of memory cells. Rather than changes to state, the functional programming view is that computation is a sequence of transformations. Instead of changing existing values, new values are constructed which possess certain desired attributes.

These characteristics are exhibited by the functions in Figure 2.5, which illustrate a functional approach to the insertion sort algorithm. The emphasis on transformation is illustrated by the first function, which takes as argument a single value and an existing list (presumably already sorted) and returns a new list in which the value has been correctly placed so as to preserve ordering. Compare this function with the similarly named function in Figure 2.4. The most obvious difference is that the earlier function returned the result through a pass-by-reference parameter, while the latter version returns the new list as the result of executing the function. Because of this, the two actions required in the general case (inserting the element into the tail of the list, and inserting the head of the list into the resulting value) can be performed in one expression without the need to introduce a temporary variable.

The tool-building characteristic of functional programming is exemplified by the second function shown in Figure 2.5. This function is, for historical reasons, named `reduce`. The `reduce` function has nothing to do with sorting or ordering values, and thus can be used in a wide variety of situations (as we will eventually see in Chapter 14). The `reduce` function takes as arguments a list, a value named the "identity", and a binary function. If the list is empty the identity value is returned. Otherwise, the head of the list is stripped off, and the binary function applied to this value and the result of recursively applying `reduce` to the remainder of the list.

To sort a list, the `reduce` function is simply invoked using the original (unsorted) list, an empty list (represented by the undefined value `NIL`) as an identity, and the previously defined function named `insertion` as the binary function argument. The result sorts a list of values through a sequence of recursive calls on `reduce`, followed by a sequence of calls on `insertion` as the recursive calls on `reduce` are "unwound". We can envision this process

```
function insertion (x : integer, aList : intList)->intList;
begin
    if empty(aList) then
        return intList(x, NIL)
    else
        if x < head(aList) then
            return intList(x, aList)
        else
            return intList(head(aList), insertion(x, tail(aList)));
end;

function reduce (aList : intList, ident : intList,
    binFun: function(integer, intList)->intList )->intList;
begin
    if empty(aList) then
        return ident
    else
        return binFun(head(aList), reduce(tail(aList), ident, binFun));
end;

function sort (aList:intList)->intList;
begin
    return reduce(aList, NIL, insertion);
end;
```

Figure 2.5: Functional Program for Insertion Sort

working on a five-element list as follows:

```
reduce(2 5 1 4 7)
insert(2, reduce(5 1 4 7))
insert(2, insert(5, reduce(1 4 7)))
insert(2, insert(5, insert(1, reduce(4 7))))
insert(2, insert(5, insert(1, insert(4, reduce(7)))))
insert(2, insert(5, insert(1, insert(4, insert(7, reduce(NIL))))))
insert(2, insert(5, insert(1, insert(4, insert(7, NIL)))))
insert(2, insert(5, insert(1, insert(4, 7))))
insert(2, insert(5, insert(1, 4 7)))
insert(2, insert(5, 1 4 7))
insert(2, 1 4 5 7)
1 2 4 5 7
```

2.4.1 Parameterized Types

We can greatly extend the power of the tool represented by the function named `reduce` through the use of a feature in Leda called parameterized types. Functions (as well as class descriptions) can be parameterized by arguments representing types, as well as by arguments representing values. Type parameters are given in a separate list both in the function definition and in function invocation. Within the body of a parameterized type declaration, the names associated with type parameters can be used just as if they were type expressions.

A parameterized version of the `reduce` function can be given as follows:

```
function reduce [T : object]
    (aList : intList, ident : T, binFun: function(integer, T)->T )->T;
begin
    if empty(aList) then
        return ident
    else
        return binFun(head(aList), reduce[T](tail(aList), ident, binFun));
end;
```

The square bracket following the function name indicates the first argument list represents type parameters, while the second list represents value parameters. Rather than being restricted to being an integer list, the identity is now specified merely as being a value of type `T`, whatever `T` might be. (`T` is constrained to be a subclass of class `object`. However, since in Leda all classes are subclasses of class `object`, `T` can be any type whatsoever.) Similarly, the binary function takes as arguments an integer value, and a value of type `T`, and must yield a value of type `T`. Finally, since the `reduce` function returns the value yielded by the binary function, the `reduce` function itself must return a value of type `T`.

The call on the `reduce` function given in the sorting algorithm shown in Figure 2.5 must now be changed so that it specifies both a type parameter and the value parameters to be used with the call.

```
function sort (aList:intList)->intList;
begin
    return reduce [intList] (aList, NIL, insertion);
end;
```

The type parameter indicates that `T` should be replaced by the type `intList` for the purposes of this call.

The advantage of this new form of `reduce` is that it can now be used with a much wider variety of binary functions, not all of which return list values. For example, the expression `integer.plus` yields the integer binary addition operator in Leda. Using this as argument for a reduction, the following expression can be used to generate the arithmetic sum of an integer list:

```
    theSum := reduce [integer] (aList, 0, integer.plus);
```

Many other functions can be similarly generated using `reduce`. We will explore this topic in more detail later in Chapter 14.

2.5 An Object-Oriented Solution

The last approach we will examine in this chapter is the object-oriented paradigm. As we noted earlier, the object-oriented philosophy views computation as the creation of a universe populated by collections of processing elements. Each processing element, like the computer itself, consists of a collection of memory values and a series of functions that can be invoked in order to get the processing element to perform certain actions. The functions defined within the scope of a class definition are sometimes referred to as *methods*, a term we will use interchangeably with "function" in the remainder of the text.

In the object-oriented world it is considered improper to do something "to" a value (meaning, to directly modify the internal state of an object), rather, all changes to the state of an object should be performed by the object itself in response to a request (generated through an invocation of an object method). It is as if an object were a sentient being, and indeed, personification and anthropomorphic descriptions of computation abound in the literature on object-oriented programming.

These ideas are expressed in our object-oriented version of insertion sort, shown in Figure 2.6. In order to provide behavior to our data structures we introduce a new data type,

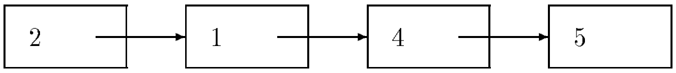
```
class orderedList;
var
    value : integer;
    next : orderedList;

    function append (newLink : orderedList);
    begin
        next := newLink;
    end;

    function initialize ()->orderedList;
    var
        n : orderedList;
    begin
        if defined(self.next) then
            if next.value < self.value then begin
                n := self.next;
                self.next := n.next;
                n.append(self.initialize());
                return n;
            end;
        return self;
    end;
end;

function sort (aList : intList)->orderedList;
var
    newList : orderedList;
begin
    while defined(aList) do begin
        newList := orderedList(head(aList), newList);
        newList := newList.initialize();
        aList := tail(aList);
    end;
    return newList;
end;
```

Figure 2.6: An Object-Oriented Insertion Sort

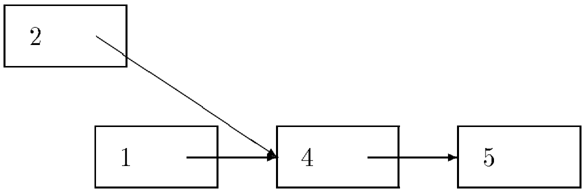
called an ordered list. In addition to maintaining two data values, an ordered list has two actions that it can perform. The behavior for instances of a class is indicated by the functions defined within the scope of the class description. The first action is named "append". It merely changes the value of the next-link data field in the structure. (Remember, it is not considered proper to directly modify the data fields in an object; rather, you must request, through a function call, that the object modify itself). The second action is more complex, and is at the heart of the insertion sorting algorithm. This action requests that a value initialize itself, by checking to see that it is in order, changing the list, if necessary, to ensure it is ordered.

To form a sorted list each element will be added one by one. As each new element is added to the front of a list, the element is asked (using the initialize function) to place itself in the proper relationship to the other elements.

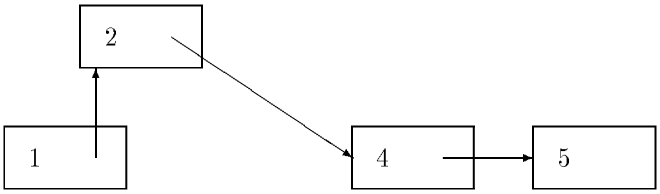
To see how this will work, consider the task of adding the value 2 to the existing ordered list consisting of the elements 1, 4 and 5. First, a new ordered list is generated (using the orderedList constructor) which places the value on the front of the list.



Next, the first element is asked to initialize itself. In doing so, the element discovers that it is out of order. It thus changes its own next field to point to the next field of its neighbor, then recursively calls initialize on itself.



In this case the recursive call will have no effect, since the element is now properly sorted. It therefore asks the former second element (still pointed to by the variable `n`) to change its next pointer to the result of the recursive call. The variable pointer to by `n` is thus the new head of the list, and is returned.



The most important characteristic to note concerning the object-oriented solution is that information is encapsulated within the memory of individual objects, and furthermore no *changes* to the state of these objects are ever made except by functions associated with the objects themselves. There are two major reasons why this is important:

- It means that to understand how a class is being used it is sufficient to understand the effect of each of the functions that manipulate the class.

- By reducing the connections between a class and a larger program to just a sequence of function calls, it is easier to move an entire class declaration from one project to the next. Thus, like the functional programming approach, object-oriented programming facilitates and encourages the development of reusable general-purpose software tools.

2.6 Comparisons / Combinations

It is useful to reiterate the different philosophical approaches embodied in the various sorting algorithms.

- Imperative programs emphasize changes to the state of memory. Memory is modified a little bit at a time, until by a sequence of small changes the desired result is obtained. Invariants can often be usefully employed to describe the state of the computation at a given point in execution; that is, the amount of work that has been done up to the given point, and the amount of work remaining to be done.

- The logical paradigm is descriptive in nature. The programmer defines the characteristics of the desired solution. By restricting this description to a specific form, the description can then be executed in a manner that results in a value being produced that satisfies the desired properties.

- The functional paradigm emphasizes transformation of values into new values, instead of changes to the state of values (as in the imperative paradigm). In addition, as with the object-oriented paradigm, the functional paradigm encourages the development of general-purpose reusable software tools.

- The object-oriented paradigm emphasizes encapsulation, information-hiding, and the solution of problems through the assignment of behavior to a universe of individual computing agents. Programming consists of writing the correct behavioral specifications ("job descriptions") for each individual computing agent, then setting them in motion.

The multiplicity of solutions in part reflects the diversity in human problem solving techniques. For some problems it seems most natural to focus on tranformation (a functional approach), whereas other times it may seem equally natural to focus on the data values themselves and their changes over time (imperative or object-oriented), whereas yet other times the primary focus may seem to be the task of searching (the logic programming approach). Multiparadigm programming provides the programmer with the ability to select a focus most apprpriate to the problem at hand, and to select a programming mechanism most appropriate to the chosen method of analysis.

It is tempting to try to compare the various solutions we have presented, but it can also be misleading. The danger lies in the natural inclination to draw unwarranted conclusions from an extremely small set of data points. For example, one might observe that the logic program version seems to be the shortest of the four, and thereby conclude that logic programs are always shorter than their imperative or functional cousins. But such a conclusion is neither true nor justified by the evidence. Similar comments can be made concerning attributes such as legibility or execution time performance.

Rather than trying to divisively pit one paradigm (or one school of programming) against another, the primary thesis behind the design of Leda is that each paradigm has strengths and weaknesses; each paradigm can be made to shine in the right circumstances. Elegant and powerful solutions to problems in computing can be found by bringing all the features of the different paradigms to bear on a given problem. We will explore, in the next Chapter, one such application.

Exercises

1. Despite the caution as to the limitations in the use of such figures, describe how a programmer might obtain a table of the actual running time for each of the four algorithms. Then, gather these figures.

2. Are the figures you collected in the previous question influenced at all by the ordering of the initial data values? What happens if the initial values are in sequence? What happens if they are in reverse sequence?

3. Quicksort is another classic sorting algorithm that can be easily described in a logic programming style. Start by defining the characteristics of a list that represents a partition of a vector. Then, describe the characteristics of quicksort in terms of partitions.

4. Describe how to compute the product of a list of integers using `reduce`. Can you think of other uses for reduce?

5. The object-oriented sorting algorithm relies on the fact that items will be checked each time a new element is inserted at the head of the list. An alternative approach would construct the entire line, then begin sorting by passing a message to the first element. Provide an example list to show that the given algorithm will not work in this situation, then show how the `initialize` procedure can be modified so that it will work in the new manner.